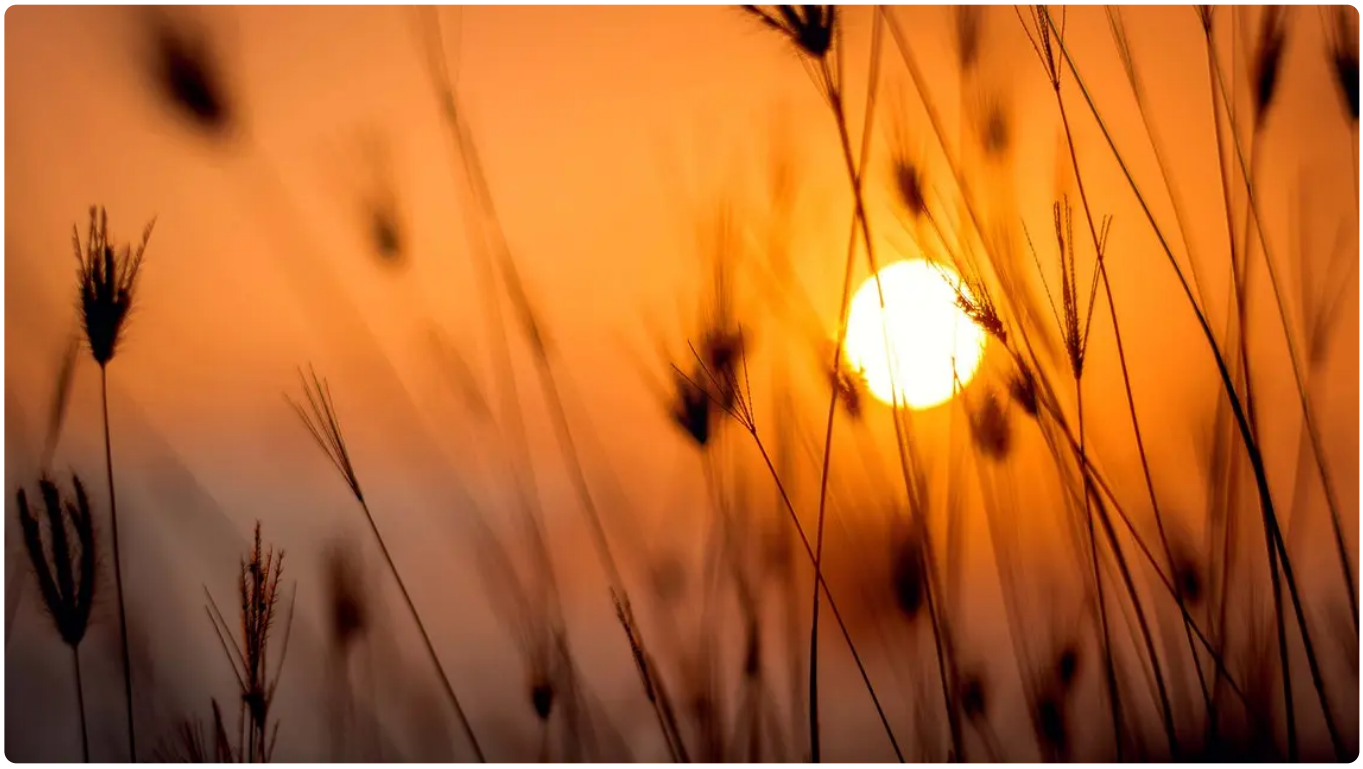
05 | 编程接口：eBPF 程序是怎么跟内核进行交互的？

2022-01-26 倪朋飞

《eBPF 核心技术与实战》 课程介绍 >



讲述：山荣
时长 12:26 大小 11.40M

你好，我是倪朋飞。

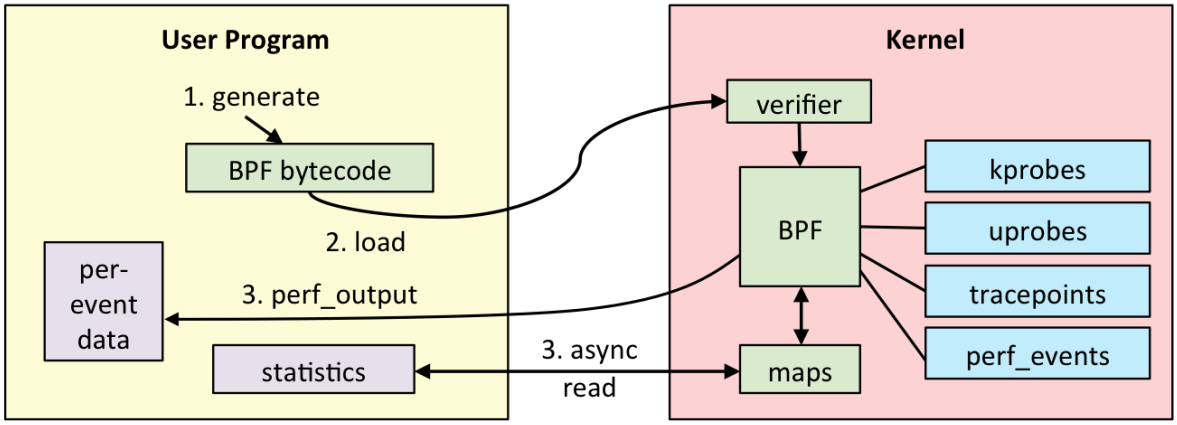
上一讲，我带你一起梳理了 eBPF 在内核中的实现原理，以及 BPF 指令的具体格式。

用高级语言开发的 eBPF 程序，需要首先编译为 BPF 字节码，然后借助 bpf 系统调用加载到内核中，最后再通过性能监控等接口与具体的内核事件进行绑定。这样，内核的性能监控模块才会在内核事件发生时，自动执行我们开发的 eBPF 程序。

那么，eBPF 程序到底是如何跟内核事件进行绑定的？又该如何跟内核中的其他模块进行交互呢？今天，我就带你一起看看 eBPF 程序的编程接口。

BPF 系统调用

如下图（图片来自brendangregg.com）所示，一个完整的 eBPF 程序通常包含用户态和内核态两部分。其中，用户态负责 eBPF 程序的加载、事件绑定以及 eBPF 程序运行结果的汇总输出；内核态运行在 eBPF 虚拟机中，负责定制和控制系统的运行状态。



对于用户态程序来说，我想你已经了解，**它们与内核进行交互时必须要通过系统调用来完成**。而对应到 eBPF 程序中，我们最常用到的就是 bpf 系统调用。

在命令行中输入 `man bpf` ，就可以查询到 BPF 系统调用的调用格式：

复制代码

```
#include <linux/bpf.h>

int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

BPF 系统调用接受三个参数：

第一个，cmd ，代表操作命令，比如上一讲中我们看到的 BPF_PROG_LOAD 就是加载 eBPF 程序；

第二个，attr，代表 bpf_attr 类型的 eBPF 属性指针，不同类型的操作命令需要传入不同的属性参数；

第三个，size ，代表属性的大小。

注意，**不同版本的内核所支持的 BPF 命令是不同的**，具体支持的命令列表可以参考内核头文件 include/uapi/linux/bpf.h 中 `bpf_cmd` 的定义。比如，v5.13 内核已经支持 36

个 BPF 命令：

复制代码

```
enum bpf_cmd {
  BPF_MAP_CREATE,
  BPF_MAP_LOOKUP_ELEM,
  BPF_MAP_UPDATE_ELEM,
  BPF_MAP_DELETE_ELEM,
  BPF_MAP_GET_NEXT_KEY,
  BPF_PROG_LOAD,
  BPF_OBJ_PIN,
  BPF_OBJ_GET,
  BPF_PROG_ATTACH,
  BPF_PROG_DETACH,
  BPF_PROG_TEST_RUN,
  BPF_PROG_GET_NEXT_ID,
  BPF_MAP_GET_NEXT_ID,
  BPF_PROG_GET_FD_BY_ID,
  BPF_MAP_GET_FD_BY_ID,
  BPF_OBJ_GET_INFO_BY_FD,
  BPF_PROG_QUERY,
  BPF_RAW_TRACEPOINT_OPEN,
  BPF_BTF_LOAD,
  BPF_BTF_GET_FD_BY_ID,
  BPF_TASK_FD_QUERY,
  BPF_MAP_LOOKUP_AND_DELETE_ELEM,
  BPF_MAP_FREEZE,
  BPF_BTF_GET_NEXT_ID,
  BPF_MAP_LOOKUP_BATCH,
  BPF_MAP_LOOKUP_AND_DELETE_BATCH,
  BPF_MAP_UPDATE_BATCH,
  BPF_MAP_DELETE_BATCH,
  BPF_LINK_CREATE,
  BPF_LINK_UPDATE,
  BPF_LINK_GET_FD_BY_ID,
  BPF_LINK_GET_NEXT_ID,
  BPF_ENABLE_STATS,
  BPF_ITER_CREATE,
  BPF_LINK_DETACH,
  BPF_PROG_BIND_MAP,
};
```

为了方便你掌握，我把用户程序中常用的命令整理成了一个表格，你可以在需要时参考：

BPF 命令	功能描述
BPF_MAP_CREATE	创建一个 BPF 映射
BPF_MAP_LOOKUP_ELEM BPF_MAP_UPDATE_ELEM BPF_MAP_DELETE_ELEM BPF_MAP_LOOKUP_AND_DELETE_ELEM BPF_MAP_GET_NEXT_KEY	BPF 映射相关的操作命令，包括查找、更新、删除以及遍历等
BPF_PROG_LOAD	验证并加载 BPF 程序
BPF_PROG_ATTACH BPF_PROG_DETACH	把 BPF 程序挂载到内核事件上 把 BPF 程序从内核事件上卸载
BPF_OBJ_PIN	把 BPF 程序或映射挂载到sysfs中的/sys/fs/bpf目录中 （常用于保持 BPF 程序在内核中贮存）
BPF_OBJ_GET	从/sys/fs/bpf目录中查找 BPF 程序
BPF_BTF_LOAD	验证并加载 BTF 信息

极客时间

BPF 辅助函数

说完用户态程序的 bpf 系统调用格式，我们再来看看内核态的 eBPF 程序。

eBPF 程序并不能随意调用内核函数，因此，内核定义了一系列的辅助函数，用于 eBPF 程序与内核其他模块进行交互。比如，上一讲的 Hello World 示例中使用的 bpf_trace_printk() 就是最常用的一个辅助函数，用于向调试文件系统（/sys/kernel/debug/tracing/trace_pipe）写入调试信息。

这里补充一个知识点：从内核 5.13 版本开始，部分内核函数（如 `tcp_slow_start()`、`tcp_reno_ssthresh()` 等）也可以被 BPF 程序直接调用了，具体你可以查看这个链接。 不过，这些函数只能在 TCP 拥塞控制算法的 BPF 程序中调用，所以本课程不会过多展开。

需要注意的是，并不是所有的辅助函数都可以在 eBPF 程序中随意使用，不同类型的 eBPF 程序所支持的辅助函数是不同的。比如，对于 Hello World 示例这类内核探针（kprobe）类型的 eBPF 程序，你可以在命令行中执行 `bpftool feature probe` ，来查询当前系统支持的辅助函数列表：

复制代码

```
1 $ bpftool feature probe
2 ...
```

```
eBPF helpers supported for program type kprobe:
  - bpf_map_lookup_elem
  - bpf_map_update_elem
  - bpf_map_delete_elem
  - bpf_probe_read
  - bpf_ktime_get_ns
  - bpf_get_prandom_u32
  - bpf_get_smp_processor_id
  - bpf_tail_call
  - bpf_get_current_pid_tgid
  - bpf_get_current_uid_gid
  - bpf_get_current_comm
  - bpf_perf_event_read
  - bpf_perf_event_output
  - bpf_get_stackid
  - bpf_get_current_task
  - bpf_current_task_under_cgroup
  - bpf_get_numa_node_id
  - bpf_probe_read_str
  - bpf_perf_event_read_value
  - bpf_override_return
  - bpf_get_stack
  - bpf_get_current_cgroup_id
  - bpf_map_push_elem
  - bpf_map_pop_elem
  - bpf_map_peek_elem
  - bpf_send_signal
  - bpf_probe_read_user
  - bpf_probe_read_kernel
  - bpf_probe_read_user_str
  - bpf_probe_read_kernel_str
...
```

对于这些辅助函数的详细定义，你可以在命令行中执行 `man bpf-helpers` ，或者参考内核头文件 include/uapi/linux/bpf.h ，来查看它们的详细定义和使用说明。为了方便你掌握，我把常用的辅助函数整理成了一个表格，你可以在需要时参考：

辅助函数	功能描述
bpf_trace_printk(fmt, fmt_size, ...)	向调试文件系统写入调试信息
bpf_map_lookup_elem(map, key) bpf_map_update_elem(map, key, value, flags) bpf_map_delete_elem(map, key)	BPF映射操作函数，分别是查找、更新和删除元素
bpf_probe_read(dst, size, ptr) bpf_probe_read_user(dst, size, ptr) bpf_probe_read_kernel(dst, size, ptr)	从内存指针中读取数据 从用户空间内存指针中读取数据 从内核空间内存指针中读取数据
bpf_probe_read_str(dst, size, ptr) bpf_probe_read_user_str(dst, size, ptr) bpf_probe_read_kernel_str(dst, size, ptr)	从内存指针中读取字符串 从用户空间内存指针中读取字符串 从内核空间内存指针中读取字符串
bpf_ktime_get_ns()	获取系统启动以来的时长，单位纳秒
bpf_get_current_pid_tgid()	获取当前线程的TGID(高32位)和PID(低32位)
bpf_get_current_comm(buf, size)	获取当前线程的任务名称
bpf_get_current_task()	获取当前任务的task结构体
bpf_perf_event_output(ctx, map, flags, data, size)	向性能事件缓冲区中写入数据
bpf_get_stackid(ctx, map, flags)	获取内核态和用户态调用栈

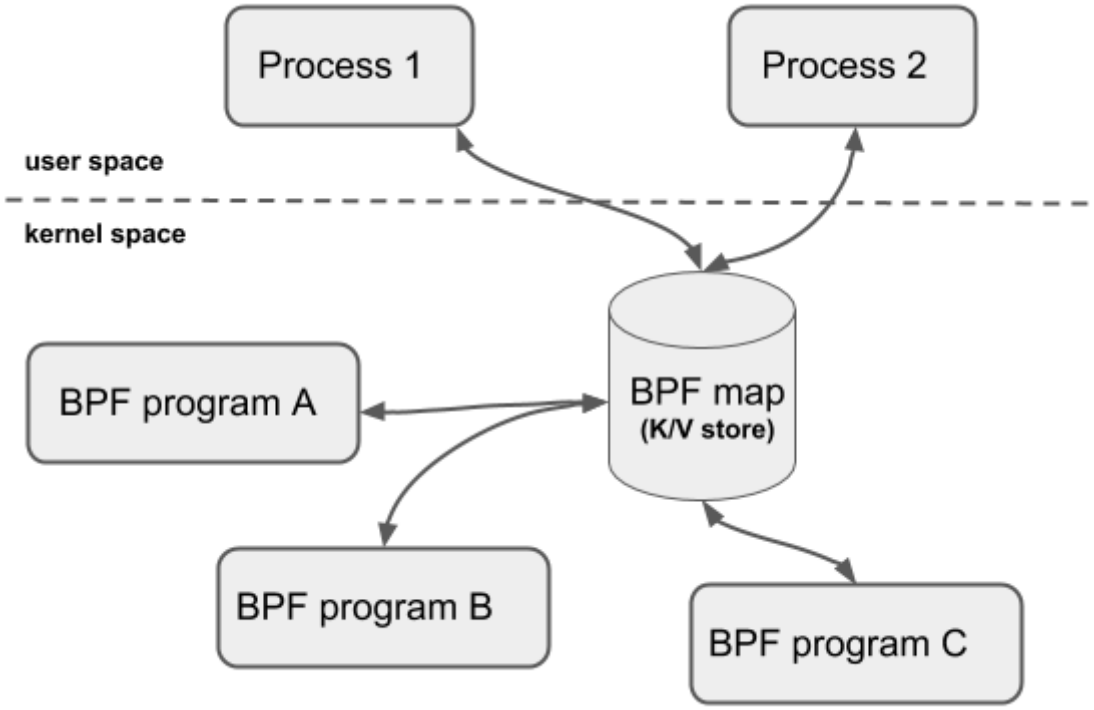
极客时间

这其中，需要你特别注意的是以bpf_probe_read 开头的一系列函数。我在上一讲中已经提到，eBPF 内部的内存空间只有寄存器和栈。所以，要访问其他的内核空间或用户空间地址，就需要借助 bpf_probe_read 这一系列的辅助函数。这些函数会进行安全性检查，并禁止缺页中断的发生。

而在 eBPF 程序需要大块存储时，就不能像常规的内核代码那样去直接分配内存了，而是必须通过 BPF 映射（BPF Map）来完成。接下来，我带你看看 BPF 映射的具体原理。

BPF 映射

BPF 映射用于提供大块的键值存储，这些存储可被用户空间程序访问，进而获取 eBPF 程序的运行状态。eBPF 程序最多可以访问 64 个不同的 BPF 映射，并且不同的 eBPF 程序也可以通过相同的 BPF 映射来共享它们的状态。下图（图片来自 docs.cilium.io ）展示了 BPF 映射的基本使用方法。



BPF 映射

在前面的 BPF 系统调用和辅助函数小节中，你也看到，有很多系统调用命令和辅助函数都是用来访问 BPF 映射的。我相信细心的你已经发现了：BPF 辅助函数中并没有 BPF 映射的创建函数，BPF 映射只能通过用户态程序的系统调用来创建。比如，你可以通过下面的示例代码来创建一个 BPF 映射，并返回映射的文件描述符：

复制代码

```
int bpf_create_map(enum bpf_map_type map_type,
        unsigned int key_size,
        unsigned int value_size, unsigned int max_entries)
{
  union bpf_attr attr = {
    .map_type = map_type,
    .key_size = key_size,
    .value_size = value_size,
    .max_entries = max_entries
  };
  return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
}
```

这其中，最关键的是设置映射的类型。内核头文件 include/uapi/linux/bpf.h 中的 bpf_map_type 定义了所有支持的映射类型，你可以使用如下的 bpftool 命令，来查询当前系统支持哪些映射类型：

复制代码

```
$ bpftool feature probe | grep map_type
eBPF map_type hash is available
eBPF map_type array is available
eBPF map_type prog_array is available
eBPF map_type perf_event_array is available
eBPF map_type percpu_hash is available
eBPF map_type percpu_array is available
eBPF map_type stack_trace is available
...
```

在下面的表格中，我给你整理了几种最常用的映射类型及其功能和使用场景：

映射类型	功能描述
BPF_MAP_TYPE_HASH	哈希表映射，用于保存 key/value 对
BPF_MAP_TYPE_LRU_HASH	类似于哈希表映射，但在表满的时候自动按LRU 算法删除最久未被使用的元素
BPF_MAP_TYPE_ARRAY	数组映射，用于保存固定大小的数组（注意数组元素无法删除）
BPF_MAP_TYPE_PROG_ARRAY	程序数组映射，用于保存 BPF 程序的引用，特别适合于尾调用（即调用其他 eBPF 程序）
BPF_MAP_TYPE_PERF_EVENT_ARRAY	性能事件数组映射，用于保存性能事件跟踪记录
BPF_MAP_TYPE_PERCPU_HASH BPF_MAP_TYPE_PERCPU_ARRAY	每个 CPU 单独维护的哈希表和数组映射
BPF_MAP_TYPE_STACK_TRACE	调用栈跟踪映射，用于存储调用栈信息
BPF_MAP_TYPE_ARRAY_OF_MAPS BPF_MAP_TYPE_HASH_OF_MAPS	映射数组和映射哈希，用于保存其他映射的引用
BPF_MAP_TYPE_CGROUP_ARRAY	CGROUP 数组映射，用于存储 cgroups 引用
BPF_MAP_TYPE_SOCKMAP	套接字映射，用于存储套接字引用，特别适用于套接字重定向

极客时间

如果你的 eBPF 程序使用了 BCC 库，你还可以使用预定义的宏来简化 BPF 映射的创建过程。比如，对哈希表映射来说，BCC 定义了 `BPF_HASH(name, key_type=u64, leaf_type=u64, size=10240)`，因此，你就可以通过下面的几种方法来创建一个哈希表映射：

复制代码

```
// 使用默认参数 key_type=u64, leaf_type=u64, size=10240
BPF_HASH(stats);

// 使用自定义key类型，保持默认 leaf_type=u64, size=10240
struct key_t {
  char c[80];
```

```
};
BPF_HASH(counts, struct key_t);

// 自定义所有参数
BPF_HASH(cpu_time, uint64_t, uint64_t, 4096);
```

除了创建之外，映射的删除也需要你特别注意。BPF 系统调用中并没有删除映射的命令，这是因为 **BPF 映射会在用户态程序关闭文件描述符的时候自动删除**（即close(fd)）。如果你想在程序退出后还保留映射，就需要调用 BPF_OBJ_PIN 命令，将映射挂载到 /sys/fs/bpf 中。

在调试 BPF 映射相关的问题时，你还可以通过 bpftool 来查看或操作映射的具体内容。比如，你可以通过下面这些命令创建、更新、输出以及删除映射：

复制代码

```
//创建一个哈希表映射，并挂载到/sys/fs/bpf/stats_map(Key和Value的大小都是2字节)
bpftool map create /sys/fs/bpf/stats_map type hash key 2 value 2 entries 8 nam

//查询系统中的所有映射
bpftool map
//示例输出
//340: hash  name stats_map  flags 0x0
//        key 2B  value 2B  max_entries 8  memlock 4096B

//向哈希表映射中插入数据
bpftool map update name stats_map key 0xc1 0xc2 value 0xa1 0xa2

//查询哈希表映射中的所有数据

bpftool map dump name stats_map
//示例输出
//key: c1 c2  value: a1 a2
//Found 1 element

//删除哈希表映射
rm /sys/fs/bpf/stats_map
```

BPF 类型格式（BTF）

了解过 BPF 辅助函数和映射之后，我们再来看一个开发 eBPF 程序时最常碰到的问题：内核数据结构的定义。

在安装 BCC 工具的时候，你可能就注意到了，内核头文件 `linux-headers-$(uname -r)` 也是必须要安装的一个依赖项。这是因为 BCC 在编译 eBPF 程序时，需要从内核头文件中找到相应的内核数据结构定义。这样，你在调用 `bpf_probe_read` 时，才能从内存地址中提取到正确的数据类型。

但是，编译时依赖内核头文件也会带来很多问题。主要有这三个方面：

首先，在开发 eBPF 程序时，为了获得内核数据结构的定义，就需要引入一大堆的内核头文件；

其次，内核头文件的路径和数据结构定义在不同内核版本中很可能不同。因此，你在升级内核版本时，就会遇到找不到头文件和数据结构定义错误的问题；

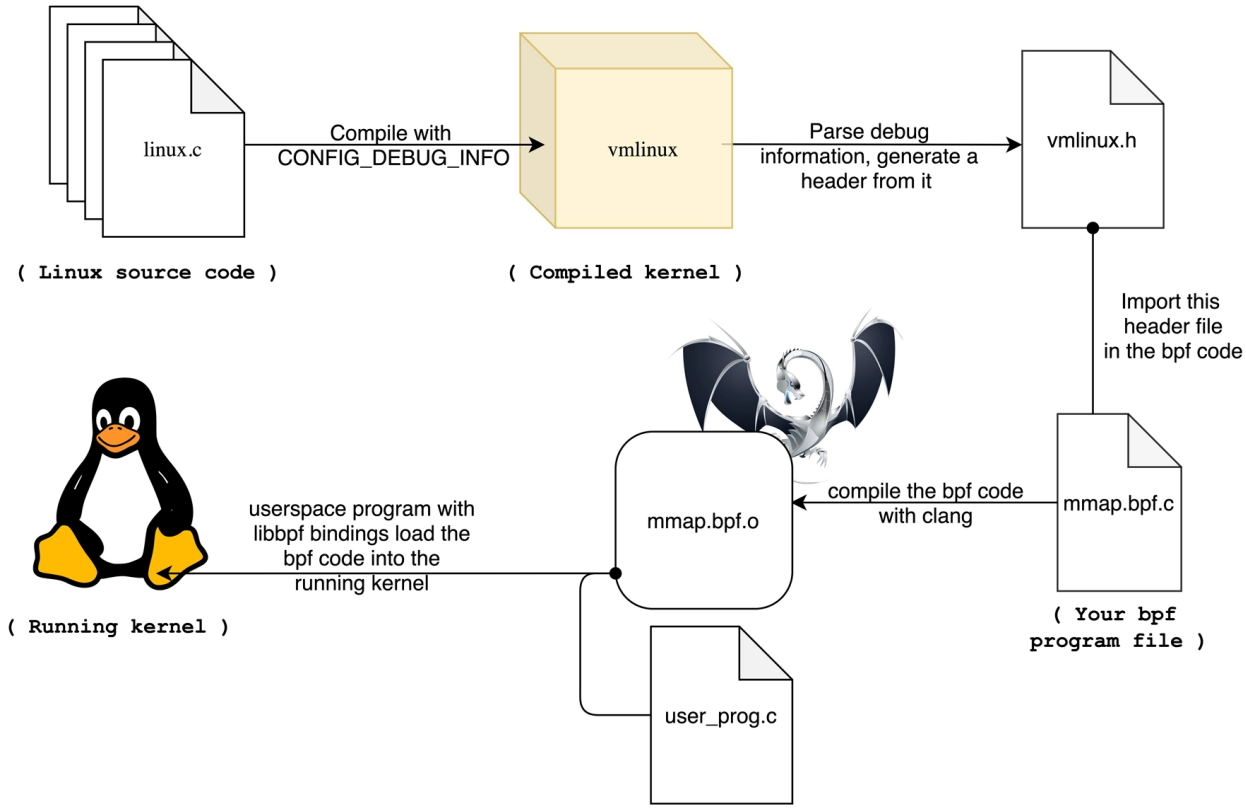
最后，在很多生产环境的机器中，出于安全考虑，并不允许安装内核头文件，这时就无法得到内核数据结构的定义。**在程序中重定义数据结构**虽然可以暂时解决这个问题，但也很容易把使用着错误数据结构的 eBPF 程序带入新版本内核中运行。

那么，这么多的问题该怎么解决呢？不用担心，BPF 类型格式（BPF Type Format, BTF）的诞生正是为了解决这些问题。从内核 5.2 开始，只要开启了 `CONFIG_DEBUG_INFO_BTF`，在编译内核时，内核数据结构的定义就会自动内嵌在内核二进制文件 vmlinux 中。并且，你还可以借助下面的命令，把这些数据结构的定义导出到一个头文件中（通常命名为 `vmlinux.h`）：

复制代码

```
1 bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

如下图（图片来自GRANT SELTZER 博客）所示，有了内核数据结构的定义，你在开发 eBPF 程序时只需要引入一个 `vmlinux.h` 即可，不用再引入一大堆的内核头文件了。



vmlinux.h的使用示意图

同时，借助 BTF、bpftool 等工具，我们也可以更好地了解 BPF 程序的内部信息，这也会让调试变得更加方便。比如，在查看 BPF 映射的内容时，你可以直接看到结构化的数据，而不只是十六进制数值：

复制代码

```
# bpftool map dump id 386
[
  {
      "key": 0,
      "value": {
          "eth0": {
              "value": 0,
              "ifindex": 0,
              "mac": []
          }
      }
  }
]
```

解决了内核数据结构的定义问题，接下来的问题就是，**如何让 eBPF 程序在内核升级之后，不需要重新编译就可以直接运行**。eBPF 的一次编译到处执行（Compile Once Run

Everywhere，简称 CO-RE）项目借助了 BTF 提供的调试信息，再通过下面的两个步骤，使得 eBPF 程序可以适配不同版本的内核：

第一，通过对 BPF 代码中的访问偏移量进行重写，解决了不同内核版本中数据结构偏移量不同的问题；

第二，在 libbpf 中预定义不同内核版本中的数据结构的修改，解决了不同内核中数据结构不兼容的问题。

BTF 和一次编译到处执行带来了很多的好处，但你也需要注意这一点：它们都要求比较新的内核版本（>=5.2），并且需要非常新的发行版（如 Ubuntu 20.10+、RHEL 8.2+ 等）才会默认打开内核配置 `CONFIG_DEBUG_INFO_BTF`。对于旧版本的内核，虽然它们不会再去内置 BTF 的支持，但开源社区正在尝试通过 BTFHub 等方法，为它们提供 BTF 调试信息。

小结

今天，我带你一起梳理了 eBPF 程序跟内核交互的基本方法。

一个完整的 eBPF 程序，通常包含用户态和内核态两部分：用户态程序需要通过 BPF 系统调用跟内核进行交互，进而完成 eBPF 程序加载、事件挂载以及映射创建和更新等任务；而在内核态中，eBPF 程序也不能任意调用内核函数，而是需要通过 BPF 辅助函数完成所需的任务。尤其是在访问内存地址的时候，必须要借助 `bpf_probe_read` 系列函数读取内存数据，以确保内存的安全和高效访问。

在 eBPF 程序需要大块存储时，我们还需要根据应用场景，引入特定类型的 BPF 映射，并借助它向用户空间的程序提供运行状态的数据。

这一讲的最后，我还带你一起了解了 BTF 和 CO-RE 项目，它们在提供轻量级调试信息的同时，还解决了跨内核版本的兼容性问题。很多开源社区的 eBPF 项目（如 BCC 等）也都在向 BTF 进行迁移。

思考题

最后，我想邀请你来聊一聊：

1. 你是如何理解 BPF 系统调用和 BPF 辅助函数的？
2. 除了今天讲到的内容，bpftool 还提供了哪些有趣的功能呢？给你一个小提示：可以使用 man bpftool 查询它的使用文档。

期待你在留言区和我讨论，也欢迎把这节课分享给你的同事、朋友。让我们一起在实战中演练，在交流中进步。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 04 | 运行原理：eBPF 是一个新的虚拟机吗？

下一篇 06 | 事件触发：各类 eBPF 程序的触发机制及其应用场景

精选留言 (5)

写留言

莫名

2022-01-26

倪老师留的思考题比较基础，尝试从另一个角度做下觉得有趣的对比：

1、bpf 系统调用一定程度上参考了 perf_event_open 设计，bpf_attr、perf_event_attr 均包含了大量 union，用于适配不同的 cmd 或者性能监控事件类型。因此，bpf、perf_event_open 接口看似简单，其实是大杂烩。...

展开

作者回复: 非常棒的思考思路👍

6

Geek_e8988c
2022-01-26

飞哥，你好，看完前两章感觉比较迷糊。没能做到学以致用，或者说举一反三。
例如第一个ebpf程序，通过trace open系统调用来监控应用的open动作。
那我实际中，有没有什么快速的方案/套路来完成一些其他的trace，例如我想知道监控那些程序使用了socket/bind(尤其适用于有些短暂进程使用udp发送了一些报文就立马退出了）。...
展开

作者回复: 非常欣慰你在看到一个入门示例就能思考这么多的问题。基础入门篇的内容主要侧重在eBPF 本身，而其应用的方法都包含在实战进阶篇（其实看看目录就知道这些问题都包含在实战进阶篇中）。

不过既然问到了，我就提前先给你介绍一个方便的工具 https://github.com/iovisor/bpftrace。想你问到的查询跟踪点函数、查询系统调用格式、快速跟踪一个系统调用、网络socket的跟踪等等它都是支持的，并且用起来就像SHELL脚本一样方便。你可以先学习一下，我们后面的案例中还会讲到它的详细使用方法。

2

Geek_59a6f9
2022-02-04

在高版本内核编译运行的ebpf程序，移植到低版本也能直接运行吗？低版本的libbpf 如何感知到高版本内核的修改，从而预定义不同内核版本的数据结构吗？

作者回复: 从低版本到高版本的兼容性可以用CO-RE解决（CO-RE只在新版本内核才支持），而反过来就需要在eBPF内部加上额外的代码来处理了。比如根据内核版本作条件编译，低版本内核执行不同的逻辑。

写点啥呢
2022-01-29

想请教下老师关于bpf内核态程序的成名周期，比如前几节课的例子，如果我通过ctrl-c终端用户态程序的时候在bpf虚拟机会发生什么呢？我理解应该会结束已经加载的bpf指令，清理map之类的资源等等，具体会有哪些，这些操作是如何在我们程序中自动插入和实现，请老师指点下。
展开

作者回复: 非常好的问题。ctrl-c其实终止的是用户空间的程序，终止后所有的文件描述符都会释放掉，对应的map和ebpf程序引用都会减到0，进而它们也就被自动回收了。这些都是自动的，不需要在程序里面增加额外的处理逻辑。

当然，在需要长久保持eBPF程序的情境中（比如XDP和TC程序），eBPF程序和map的生命周期是可以在程序内控制的，比如通过 tc、ip 以及 bpftool 等工具（当然它们也都有相应的系统调用）。我们课程后面还会讲到相关的案例。

不了峰
2022-01-28

BPF 系统调用 --> 发生在 用户态
BPF 辅助函数 --> 内核态
---
root@ubuntu-impish:/proc# bpftool perf
pid 38110 fd 6: prog_id 572 kprobe func do_sys_openat2 offset 0…
展开

作者回复: 不错的总结

1